```
// Computer Program Listing Appendix Under 37 CFR 1.52(e)
// Code.txt
// Copyright (c) 2004. Sybase, Inc. All Rights Reserved.
/**
 * The following class represents a clone of an operator
 * This is the smallest schedulable/ runnable unit in a
 * parallel query plan
 */
class OpClone
{
public:
 /**
  * Ctor
  *
  * @param phdr - ptr to the OptPool
  * @param node - the pop whose clone is being created
  * @param numtsdim - max arity of PS resource
  * @param numssdim - max arity of NPS resource
  */
 SYB_INLINE
 OpClone(
 OptPool  *phdr,
 Ppop  *node,
 int32  numtsdim,
 int32  numssdim);
/**
 * Overloaded operators are used to sort the clones
 * based on their ids
 */
SYB_INLINE bool operator<( const OpClone& second) const;
/**
 * CloneGetNpsResource
 * Get the NpS resource for this clone
 *
 * @return the Nps resource
 */
SYB_INLINE NpsResource *
CloneGetTsResource(void) const;
/**
 * CloneGetPsResource
 * Get the PS resource for this clone
 *
 * @return the PS resource
 */
SYB_INLINE NpsResource *
CloneGetNpsResource(void) const;
/**
 * CloneSetSiteId
 * Set the site id to where the clone will be
  * executing
  *
 * @param siteid - the site id for the clone
 */
SYB_INLINE void
CloneSetSiteId(SiteIdType siteid);
/**
 * CloneGetSiteId
 * Get the site id for this clone
 *
 * @return the site id for the clone
 */
SYB_INLINE SiteIdType
CloneGetSiteId(void) const;
/**
 * CloneGetPop
 * Get the pop to which this clone belong to
 *
 * @return the pop
```

```
   */
  SYB_INLINE Ppop *
  CloneGetPop(void) const;
private:
  /*
   * The Pop node that its is cloned from
   */
  Ppop  *_cloneOfPop;
  /*
   * NPS resource for the clone
   */
  NpsResource *_npsUsage;
  /*
   * PS resource for the clone
   */
  PsResource *_psUsage;
  /*
   * Id of the clone, based on the sequence of partitions
   */
  CloneIdType _cloneId;
  /*
   * the site id where this clone needs to be run
   */
  SiteIdType _siteId;
};
/**
 * The concept of a macro-clone is rather bizzare. We came up
 * with this to reflect that clones are really not "floating"
 * after all. They are floating only to an extent such that clones
 * that participate in a legal relational operation needs to be
 * local on a given site to minimize data movement.
 * If two clones from two different operators are not broken by an
 * Xchg, then there must be some benefit in running them at the same
 * site. Consider the following physical tree
 *     xchg
 *      /
 *    join-A( 2 partitions)
 *     /
 *   join-B ( 2 partitions)
 *    /
 * xchg
 *
 * We may want to keep the corresponding clones of the two joins
 * on one site. Otherwise we shall have to do a data movement
 * which had not been accounted for in the core optimizer.
 */
class MacroClone
{
public:
SYB_INLINE
MacroClone(void);
/**
 * Ctor
 *
 * @param phdr - ptr to memory pool
 * @param clone - ptr to a clone
 */
SYB_INLINE
MacroClone(
 OptPool  *phdr,
 OpClone  *clone);
SYB_INLINE const OpCloneListType &
McGetCloneSet(void) const;
/**
 * standard comparison operator less than is overloaded
 * because we have a need to sort the macroclones
 *
 * @return TRUE if less, FALSE, otherwise
```

```
 */
SYB_INLINE bool operator<( const MacroClone& second) const;
/**
 * McComputeNpsResource
 * This computes the NPS resource for a macro-clone
 */
SYB_INLINE void
McComputeNpsResource();
/**
 * McComputePsResource
 * This computes the PS resource for a macro-clone
 */
SYB_INLINE void
McComputePsResource();
/**
 * McGetPsResource
 *  A macro clone can be asked for Ps resource, which would
 * be a union of all its clones' Ps resources. This information
 * is usually cached.
 *
 * @return the PsResource that is a summation of PsResource
 * for all clones
 */
SYB_INLINE PsResource *
McGetPsResource() const;
/**
 * McGetNpsResource
 *  A macro clone can be asked for Nps resource, which would
 * be a union of all its clones' Nps resources. This information
 * is usually cached.
 *
 * @return the NpsResource that is a summation of NpsResource
 * for all clones
 */
SYB_INLINE NpsResource *
McGetNpsResource() const;
/**
 * McSatisfied
 *  Check if the "avail" (available) resource is
 * sufficient enough for this macro clone to fit in. This is
 * achieved by adding up the current macroclone's resource to
 * "used" and then seeing if we've exceeded the capacity.
 * It is interesting that we use "avail" and not 1.0 as the max
 * capacity, since it takes care of heterogeneious cluster
 * (heterogeneous in terms of non-preemptable resources)
 *
 * @param avail - available non-preemptable resource
 * @param used - non-preemptable resource that has been used
 */
SYB_INLINE SYB_BOOLEAN
McSatisfied(
 NpsResource  *avail,
 NpsResource  *used);
/**
 * McAddOpClone
 * Add a new operator clone to this macro clone
 *
 * @param clone - a new operator clone to add
 */
SYB_INLINE void
McAddOpClone(OpClone *clone);
/**
 * McGetCloneCount
 *  Get the number of clones in a macro-clone
 *
 * @return the number of clones
 */
SYB_INLINE int32
```

```
McGetCloneCount(void) const;
/**
 * McSetSiteId
 *  Set site id information for each clone
 *
 * @param siteid - the site id
 */
SYB_INLINE void
McSetSiteId(SiteIdType siteid);
/**
 * McGetSiteId
 *  Get site id info for the clones; they should share the
 * same site id by definition of a macroclone
 *
 * @return the site id of this macro-clone
 */
SYB_INLINE SiteIdType
McGetSiteId(void) const;
/**
 * McIsPinnedToSite
 *  Check if a macro clone is pinned to a site or not ?
  *
  * @return TRUE if macro clone pinned to a site, FALSE,
  * otherwise
  */
 SYB_INLINE SYB_BOOLEAN
 McIsPinnedToSite(void) const;
private:
 /*
  * the set of clones in this macro-clone
  */
 OpCloneListType  _cloneset;
 /*
  * Collective resource usage is tracked for all
  * the clones
  */
 PsResource  *_psResForMac;
 NpsResource  *_npsResForMac;
 /*
  * state information
  */
 uint32   _state;
 static const uint32  MC_NPSRES_UPDATED;
 static const uint32  MC_PSRES_UPDATED;
};
/**
** A classical definition of a pipeline says that it is a given
** logical entity where all the participating operations start
** between (t, t + delta) and end between (t1, t1 + delta)
** Here we use a class to capture that concept. Using the property
** of a pipeline, we can include all operators that could be
** put in it and hence only the top most operator in a given
** tree fragment needs to be identified.
*/
class PipeLine
{
public:
 /**
  * Ctor
  *
  * @param top - is the pop that roots the pipeline
  */
 SYB_INLINE
 PipeLine(
  Ppop   *top);
 /**
  * getTop
  * Get the top pop of the pipeline
```

```
 *
 * @return the top pop
 */
SYB_INLINE Ppop *
getTop(void) const;
/**
 * setTop
 * Set the top of the pipeline to the pop passed
 * in as parameter
 *
 * @param topNode -  the pop passed in as parameter
 */
SYB_INLINE void
setTop(Ppop  *topnode);
/**
 * PipeGetMacroSet
 * Get the macro clone list in this pipe
 *
 * @return a reference ptr to the macro clone list
 */
SYB_INLINE MacroCloneListType&
PipeGetMacroSet(void);
/**
 * Standard less than operator that compares based on
 * max time to execute a pipeline
 */
SYB_INLINE bool
operator<(const PipeLine& other) const;
/**
 * PipePsMax
 *  Maximum sequential time taken by a pipeline to
 * complete. This method measures that time and returns
 * it to the caller
 *
 * @return the sequential execution time
 */
SYB_INLINE double
PipePsMax(void) const;
/**
 * PipeGetNpsResourceUsage
 *  Gets the total amount of Nps resource used by
 * this pipe.
 *
 * @param ssr - ptr to the NPS resource object that will
 * contain the total NPS resource usage
 */
SYB_INLINE void
PipeGetNpsResourceUsage(NpsResource  *ssr) const;
/**
 * getEventTreeNode
 *  Every pipe has a hosting node, which is a tree
 * node in a given event tree
 *
 * @return the event tree node that holds this pipe
 */
SYB_INLINE EventTreeNode *
getEventNode(void) const;
/**
 * setEventNode
 * Setup the hosting event tree node for this pipeline
 *
 * @param ttnode - event tree node
 */
SYB_INLINE void
setEventNode(EventTreeNode *ttnode);
/**
 * PipeSetSchedId
 *  Set the schedulable level of the pipe
```

```
 *
 * @param sched_level - set the schedulable level
 */
SYB_INLINE void
PipeSetSchedId(uint32 sched_level);
/**
 * PipeGetSchedId
 *  Get the schedulable level of the pipe
 *
 * @return the schedulable level of the pipe
 */
SYB_INLINE uint32
PipeGetSchedId(void);
/**
 * PipeGetCloneId
 * Get the starting id of the clones
 *
 * @return the id of the clones
 */
SYB_INLINE CloneIdType
PipeGetCloneId(void) const;
/**
 * PipeSetCloneId
 *  Set the current clone id that's available for
 * next use
 *
 * @param startid - the start id to assign to clones in
 * this pipe
 */
SYB_INLINE void
PipeSetCloneId(CloneIdType startid);
/**
 * PipeCreateClones
 *  Create all the requisite clones for a given pipe
 *
 * @param phdr - ptr to the memory pool
 */
void
PipeCreateClones(OptPool *pool);
/**
 * PipeCreateMacroClones
 *  Once the basic clones are created, this method
 * creates a macro clone by using the data placement constraints
 *
 * @param pool - ptr to a memory pool
 */
void
PipeCreateMacroClones(OptPool *pool);
/**
                 *  PipeBinPack
                 * Algorithm to run the bin packing method so as to
schedule all of the
                 * macro-clones in a given pipeline
                 *
                 * @param ssg - ptr to the global resource descriptor
                 * @param phdr - ptr to the memory pool for any memory
allocation
                 */
                 void
                 PipeBinPack(
                           SsgGlobalResource            *ssg,
                           OptPool                                *phdr);
 /**
  * PipeCollectClones
  *  collect all clones of a pipeline into a list
  * that is passed as a parameter
  *
  * @param macList - list of macro clones
```

```
  */
 void
 PipeCollectClones(MacroCloneListType &macList) const;
 /**
  * PipeMacCloneSort
  *  sort clones by the ratio of the length of their PS
                 *  resource cost vector to that of the length of the NP
S
                 *  resource cost vector (work density), which is
                 * really the crux of the bin-packing algorithm
  */
 void
 PipeMacCloneSort(void);
private:
 /*
  * A pop that starts the top of a pipeline
  */
 Ppop  *_top;
 /*
  * set of clones for this pipeline
  */
 OpCloneListType _cloneset;
 /*
  * set of macro clones for this pipeline that has met the
  * scheduling constraints
  */
 MacroCloneListType _macrocloneset;
 /*
  * The event tree node  hosting this pipeline
  */
 EventTreeNode *_hostingNode;
 /*
  * schedulable level for the pipe
  */
 uint32  _schedlevel;
 /*
  * this is the currently exercised id for a clone in a pipe
  */
 CloneIdType _startid;
 /*
  * id of the pipe
  */
 PipeIdType _pipeId;
};
/**
 * A Event Tree Node is a node in a event tree that roots a single pipel
ine
 * This tree is created based upon the interdependence of the pipelines
 */
class EventTreeNode
{
public:
 enum { TASKNODE_ALLOC_SIZE = 4};
 friend class EventTree;
 /**
  * Ctor
  *
  * @param pipe - pipeline that will be anchored by this
  * event tree node
  * @param phdr - ptr to the memory pool
  */
 SYB_INLINE
 EventTreeNode(
  PipeLine *pipe,
  OptPool  *phdr);
 ~EventTreeNode() {}
 /**
  * TtnGetPipe
```

```
 * Get the pipe anchored to this event tree node
 *
 * @return pipe anchored to this event tree node
 */
SYB_INLINE PipeLine *
TtnGetPipe(void) const;
/**
 * TtnGetArity
 * Get the arity of this event tree node
 *
 * @return the arity
 */
SYB_INLINE int32
TtnGetArity(void) const;
/**
 * TtnSetPipeScheduled
 * Set the state of the pipeline to indicate
 * that its scheduling has been completed
 */
SYB_INLINE void
TtnSetPipeScheduled(void);
/**
 * TtnIsPipeScheduled
 * Check if the pipeline for this event tree node
 * has been scheduled or not
 *
 * @return TRUE, if it is, FALSE, otherwise
 */
SYB_INLINE SYB_BOOLEAN
TtnIsPipeScheduled(void);
/**
 * TtnCreateClones
 * Create clones for all pops that belongs to the
 * pipe rooted at this event tree node
 *
 * @param phdr - ptr to the OptPool
 */
void
TtnCreateClones(OptPool *phdr);
/**
 * TtnFindUnschedPipes
 *  Find and collect all pipes that can be scheduled at a
 * given point
 *
 * @param ssg - ptr to the global resources
 * @param pipe_list - a list of pipes obtained from a call to
 * this method
 */
SYB_BOOLEAN
 TtnFindUnschedPipes(
   SsgGlobalResource   *ssg,
   PipeLineListType& pipe_list);
private:
 /*
  * A event node points to its pipeline
  */
 PipeLine *_pipeline;
 /*
  * Arity of the event tree node
  */
 int32  _arity;
 /*
  * Number of child node space allocated but not used
  */
 int32  _allocarity;
 /*
  * The event tree nodes that this event tree node depends on
  */
```

```
 EventTreeNode **_eventnodes;
 /*
  * state of a event tree node
  */
 uint32  _state;
 /*
  * indicates the pipe has been scheduled completely
  */
 static const uint32 PIPE_SCHEDULED;
};
/**
 * Event Trees represent dependencies of pipelines. These are N-arry tre
es
 * in general and is comprised of event tree nodes.
 */
class EventTree
{
public:
 /**
  * Ctor
  *
  * @param phdr - ptr to the memory pool
  * @param tnode - ptr to a event tree node that will be
  * the root node of the event tree
  */
SYB_INLINE
EventTree(
 OptPool  *phdr,
 EventTreeNode *tnode = NULL);
/**
 * TtSetRoot
 * Set the root event tree node of a event tree
 *
 * @param rootnode - root node of a event tree
 */
SYB_INLINE void
TtSetRoot(EventTreeNode *rootnode);
/**
 * TtGetRoot
 * Get the root event tree node of a event tree
 *
 * @return the root event tree node
 */
SYB_INLINE EventTreeNode *
TtGetRoot(void) const;
/**
 * TtCreateClones
 * Create clones of all pops in all pipes in the
 * event tree
  */
 void
 TtCreateClones(void);
private:
 /*
  * OptPool to allocate event trees
  */
 OptPool  *_eventhdr;
 /*
  * Root node of a event tree
  */
 EventTreeNode *_rootnode;
};
/**
**  Guts of the parallel system. This anchors the majority of the
**  scheduling algorithms
*/
class ParSchedule
{
```

```
public:
 /**
  * Ctor
  *
  * @param p - ptr to the top of the pop tree
  * @param optGbl - ptr to the OptGlobal
  * @param ssg - ptr to the global resources
  */
 ParSchedule(
  Ppop    *p,
  OptGlobal  *optGbl,
  SsgGlobalResource *ssg);
 /**
  * ParSetEventTree
  * Set the root event tree node in the schedule
  *
  * @param tnode - the top event tree node
  */
 SYB_INLINE void
 ParSetEventTree(EventTreeNode *tnode);
 /**
  * ParGetEventTree
  * Get the root event tree node in the schedule
  *
  * @return the root event tree node
  */
SYB_INLINE EventTree *
ParGetEventTree(void) const;
/**
 * ParGetPhdr
 * Get the memory pool ptr
 *
 * @return the memory pool
 */
SYB_INLINE OptPool *
ParGetPhdr(void) const;
/**
 * ParGetGlobalRes
 * Get the global resource class ptr
 *
 * @return the global resource class ptr
 */
SYB_INLINE SsgGlobalResource *
ParGetGlobalRes(void) const;
/**
 * ParCreateClones
 * The main driver method that creates clones for a given
 * event tree and identifies all constraints related to the
 * placement of the clones
 */
void
ParCreateClones(void);
/**
 * ParDriveScheduleToShelves
 *  The main level scheduling algorithm that schedules multiple
 * pipelines
 *
 * @param schedLevel - schedulable level, an id that is
 * assigned to pipes to suggest that ones having the
 * same id can all be scheduled together.
 */
void
ParDriveScheduleToShelves(uint32 *schedLevel);
/**
 * ParPartitionPipeList
 * Partition a list of pipelines {C1,C2,..Cn} into
 * L lists such that  P1={C1,..Ci1},P2={Ci1+1,..,Ci2}..,
 * LL={Ci(k-1) + 1,..Cn} such that the length of the set
```

```
 * Lj is <= P(1 - ORG), where P is the number of sites
 * and that Lj is maximal.
 *
 * @param setOfPipes - list of all pipes to be partitioned
 * and scheduled based on the schedulability rules
 */
void
ParPartitionPipeList(
 PipeLineListType         &setOfPipes);
/**
 * ParScheduleToShelves
 * Partition a list of pipelines {C1,C2,..Cn} into
 * k lists. Then schedule one or more pipelines at a time
 * in shelves
 *
 * @param setOfPipes - list of all pipes to be partitioned
 * and scheduled based on the schedulability rules
 * @param level - shelf level
 */
void
ParScheduleToSheleves(
  PipeLineListType& setOfPipes,
  uint32   *level);
/**
 * ParCollectClones
 *  Clone collection from pipelines that form a
 * collective unit for the purpose of scheduling
 *
 * @param listOfPipes - list of pipes from which clones
 * need to be collected
 * @param cloneList - list of macro clones to collect the
 * set
 */
void
ParCollectClones(
  const PipeLineListType &listOfPipes,
  MacroCloneListType  &cloneList) const;
/**
 * ParMarkPipeScheduled
 *  Mark Pipelines scheduled and set its schedulable levels
 *
 * @param listOfPipes - list of pipes that are getting
 * scheduled
 * @param level - level of the shelf
 */
void
ParMarkPipeScheduled(
  PipeLineListType *listOfPipes,
  uint32   level);
 /**
  * ParRestorePsCost
  *  The PS resource cost is cummulative inside the
  * search space. The process of  restoring cost vector
  * is one in which the actual PS costs of each operator
  * is calculated
  */
 void
 ParRestorePsCost();
 /**
  * ParSortPipes
  *  Sort PipeLines in order of their response times;
  * the order being non-decreasing in nature
  *
  * @param listOfPipes - list of pipes to be sorted
  */
 void
 ParSortPipes(PipeLineListType& listOfPipes);
private:
```

```
 /*
  * OptPool to allocate memory from
  */
 OptPool    *_phdr;
 /*
  * Ptr to the OptGlobal object which is central to optimizer
  */
 OptGlobal  *_optGbl;
 /*
  * The parallel pop tree being analysed here
  */
 Ppop    *_rootnode;
 /*
  * The event tree that will be generated
  */
 EventTree  *_eventtree;
 /*
  * The global resource information pertaining to sites etc.
  */
 SsgGlobalResource  *_globalnpsres;
};
/*
 * ParSchedule
 *
 *      The constructor walks the Pop tree and macro-expands
 *      an operator tree. It then identifies pipelines and
 *      their dependencies and comes up with an event tree model
 *      to capture this information. This event tree is subsequently
 * used to generate the scheduling information.
 *
 *      Parameters:
 *              p       : ptr to the root node of the operator tree
 *   optGbl : ptr to the global optimization object
 *   ssg : ptr to the object that holds global information
 *      about sites and their respective NPS resources
 *      returns:
 *              None
 *
 */
ParSchedule::ParPipeLineSchedule(
                Ppop                    *p,
                OptGlobal               *optGbl,
                SsgGlobalResource       *ssg)
{
        EventTreeNode    *tnode;
        OptPool          *phdr = optGbl->GblGetPhdr();
        uint32           sched_level = 1;
        TraceOut         &traceout = ::traceout();
        _eventtree = OPT_NEW(phdr, EventTree(phdr));
        _rootnode = p;
        _phdr = phdr;
        _optGbl = optGbl;
        _globalnpsres = ssg;
        // Build the event tree now
        _rootnode->identifyPipes(&tnode, phdr);
        ParSetEventTree(tnode);
// The PS resource cost is cummulative inside the
// search space. The process of  restoring cost vector
// is one in which the actual PS costs of each operator
// is calculated
ParRestorePsCost();
// Create operator clones based on partitioning degree of
// operator
ParCreateClones();
ParCreateMacroClones();
// Split pipelines so that none exceeds their NPS resource
// limit
ParSplitPipeLines();
```

```
        // Generate the schedule
        ParDriveScheduleToShelves(&sched_level);
}
/*
 * ParCreateClones
 *
 * The main driver method that creates clones for a given
 * task tree and identifies all constraints related to the
 * placement of the clones
 *
 *      Parameters:
 *              None
 *      Returns:
 *              None
 */
void
ParSchedule::ParCreateClones(void)
{
        _eventtree->TtCreateClones();
}
/*
 * PipeCreateMacroClones
 *
 *      This method creates macro clones by looking at the id
 * of the clones. This process creates an additional constraint
 * on placement of clones at a given site.
 *
 *      Parameters:
 *              phdr    : ptr to a OptPool to allocate memory from
 *      Returns:
 *              None
 */
void
PipeLine::PipeCreateMacroClones(OptPool  *phdr)
{
        OpCloneListIterType     ii;
        MacroCloneListIterType  jj;
        OpClone                 *temp_clone;
        MacroClone              *temp_mac;
        SiteIdType              site_id = NON_EXISTENT_SITEID;
        PipeCloneSort();
        for (ii = _cloneset.begin(); ii != _cloneset.end(); ++ii)
        {
                temp_clone = *ii;
                if (temp_clone->CloneGetSiteId() != site_id)
                {
                        // Create a macro clone and add to
                        // the set of macroclones
                        temp_mac = OPT_NEW(phdr, MacroClone(
                                                phdr,
                                                temp_clone));
                        _macrocloneset.push_back(temp_mac);
                        site_id = temp_clone->CloneGetSiteId();
                }
                else
                {
                        temp_mac->McAddOpClone(temp_clone);
                }
        }
        // If any of the macro clone has a "rooted" operator, which
        // means that a clone of this macroclone has been pinned, which
        // in turn makes the macro clone pinned too
        for (jj = _macrocloneset.begin(); jj != _macrocloneset.end(); ++
jj)
        {
                temp_mac = *jj;
                temp_mac->McCheckAndSetSiteId();
                temp_mac->McComputeNpsResource();
```

```
                temp_mac->McComputePsResource();
        }
}
/*
 * ParDriveScheduleToShelves
 *
 *      The main algorithm to handle scheduling of multiple
 * pipelines in a bushy tree (left deep tree included).
 * The idea is to traverse an event tree to identify all possible
 * pipelines that can be scheduled at a given point.
 * Given a set of pipelines {C1,..,Cn} that meet the ORG limits and
 * a set of P sites, the following algorithm calls a classical
 * bin packing routine to stack the pipelines in shelves
 * Output: A mapping of clones to sites that does not violate its
 * NPS constraints or pipeline dependencies
 *
 *      Parameters:
 *              schedulelevel   : schedulable level, an id that is
 *                                assigned to pipes to suggest that
 *                                ones having the same id can all be
 *                                scheduled together.
 *      Output:
 *              None
 */
void
ParSchedule::ParDriveScheduleToShelves(
                uint32                  *schedulelevel)
{
        PipeLineListType        set_of_pipes;
        _eventtree->TtGetRoot()->TtnFindUnschedPipes(
                                                _globalssres,
                                                set_of_pipes);
        if (!set_of_pipes.empty())
        {
                ParScheduleToShelves(
                                set_of_pipes,
                                schedulelevel);
        }
        // All pipes have been scheduled
        // This is great :-)
}
/*
 * TtnFindUnschedPipes
 *
 * This method goes through the event tree and finds out if there
 * are pipelines that becomes ready to schedule because some dependencie
s
 * may have been satisfied in the prior step.
 *
 *      Parameters:
 *              ssg             : ptr to the global resource
 *              set_of_pipes    : list of pipelines being examined
 *      Returns:
 *              TRUE indicates that a given node has already been
 *              scheduled
 *              FALSE, otherwise
 *              Also note that it has the side effect of adding
 *              pipes to the set_of_pipes list
 *
 */
SYB_BOOLEAN
EventTreeNode::TtnFindUnschedPipes(
                        SsgGlobalResource       *ssg,
                        PipeLineListType        &set_of_pipes);
/*
 * ParScheduleToShelves
 *
 * Partition a list of pipelines {C1,C2,..Cn} into k lists such
```

```
 * that  L1=<C1,..Ci1>,L2=<Ci1+1,..,Ci2>..,Lk=<Ci(k-1) + 1,..Cn>
 * such that the length of the set Lj is <= P(1 - ORG), where P
 * is the number of sites and that Lj is maximal
 *
 *      Parameters:
 *              set_of_pipes    : list of pipes
 *      Returns:
 *              None
 */
void
ParSchedule::ParScheduleToShelves(
                        PipeLineListType        &set_of_pipes,
                        uint32                  *schedlevel)
{
        PipeLineList2IterType   part_iter;
        PipeLineListType        *eachPipeList;
        PipeLine                temp_pipe;
        while (!set_of_pipes.empty())
        {
                // Sort the pipelines in non-increasing order
                // of their Tmax
                ParSortPipes(set_of_pipes);
                ParPartitionPipeList(set_of_pipes);
                // The list has been partitioned and everything looks
                // good
                for (part_iter = _partitionOfPipes.begin();
                        part_iter != _partitionOfPipes.end();
                                part_iter++)
                {
                        eachPipeList = *part_iter;
                        ParCollectFloatingClones(
                                        *eachPipeList,
                                        temp_pipe.PipeGetMacroSet());
                        // Mark all pipes that will get scheduled
                        // This is done in advance with the knowledge
                        // that BinPack will not bomb on us
                        ParMarkPipeScheduled(eachPipeList, *schedlevel);
                        temp_pipe.PipeBinPack(_globalssres, _phdr);
                        // Everything in tempPipe has been scheduled; so
                        // start with a clean slate and a new schedule l
evel
                        ++(*schedlevel);
                        temp_pipe.PipeCleanse();
                        // All that could not be scheduled will be refou
nd
                        // So delete them anyway
                        set_of_pipes.clear();
                        // Find all pipes that become available for
                        // scheduling
                        _eventtree->TtGetRoot()->TtnFindUnschedPipes(
                                                                _globals
sres,
                                                                set_of_p
ipes);
                        }
                }
        }
}
/*
 * PipeBinPack
 *      This method takes all operator clones in a given pipeline
 * and tries to schedule them based on bin packing algorithm. There
 * is a minimal sufficicency condition for a pipeline to be fully
 * schedulable.
 *
 *      Parameters:
 *              ssg     : ptr to the global resource structure
 *              phdr    : ptr to the OptPool to allocate memory
```

```
 *                          from
 *       Returns:
 *               None
 */
void
PipeLine::PipeBinPack(
                SsgGlobalResource       *ssg,
                OptPool                 *phdr)
{
        MacroCloneListIterType  iter;
        MacroClone              *mac_clone;
        PsCostUnitType          min_nps_cost;
        PsCostUnitType          nps_cost;
        int32                   cheapest_site_num;
        int32                   site_num;
        // Sort the macro clones in the pipeline according to
        // a specific ordering
        PipeMacCloneSort();
        // Initialize the NPS resource usage
        ssg->SsgNpsResInit();
        // Now pin the floating macroclones
        for (iter = _macrocloneset.begin(); iter != _macrocloneset.end()
;
                ++iter)
        {
                mac_clone = *iter;
                if (mac_clone->McIsPinnedToSite())
                {
                        // Those that have been pinned need not be
                        // considered again
                        continue;
                }
                min_nps_cost = NpsResource::TS_INFINITY;
                for (int32 i = 0; i < ssg->SsgGetNumSites(); i++)
                {
                        if (mac_clone->McSatisfied(
                                        ssg->SsgGetNpsResAvail(i),
                                        ssg->SsgGetNpsResUsed(i)))
                        {
                                if ((nps_cost =
                                        mac_clone->McGetNpsResource()->
                                                getResLength(
                                                     ssg->
                                                        SsgGetNpsResUsed
(i))) <
                                                                min_ps_c
ost)
                                {
                                        min_nps_cost = nps_cost;
                                        cheapest_site_num = i;
                                }
                        }
                }
                if (min_nps_cost == NpsResource::TS_INFINITY)
                {
                        // i.e. we have not been able to schedule this
                        // clone; this is an error
        SYB_ASSERT(FALSE);
                }
                else
                {
                        mac_clone->McSetSiteId(
                                        ssg->SsgGetSiteId(cheapest_site_
num));
                        ssg->SsgGetPsResUsed(cheapest_site_num)->
                                        setUnionVec(
                                                mac_clone->McGetPsResour
ce());
```

```
                                ssg->SsgGetNpsResUsed(cheapest_site_num)->
                                                setUnionVec(
                                                        mac_clone->McGetNpsResou
rce());
                        }
                }
        }
```